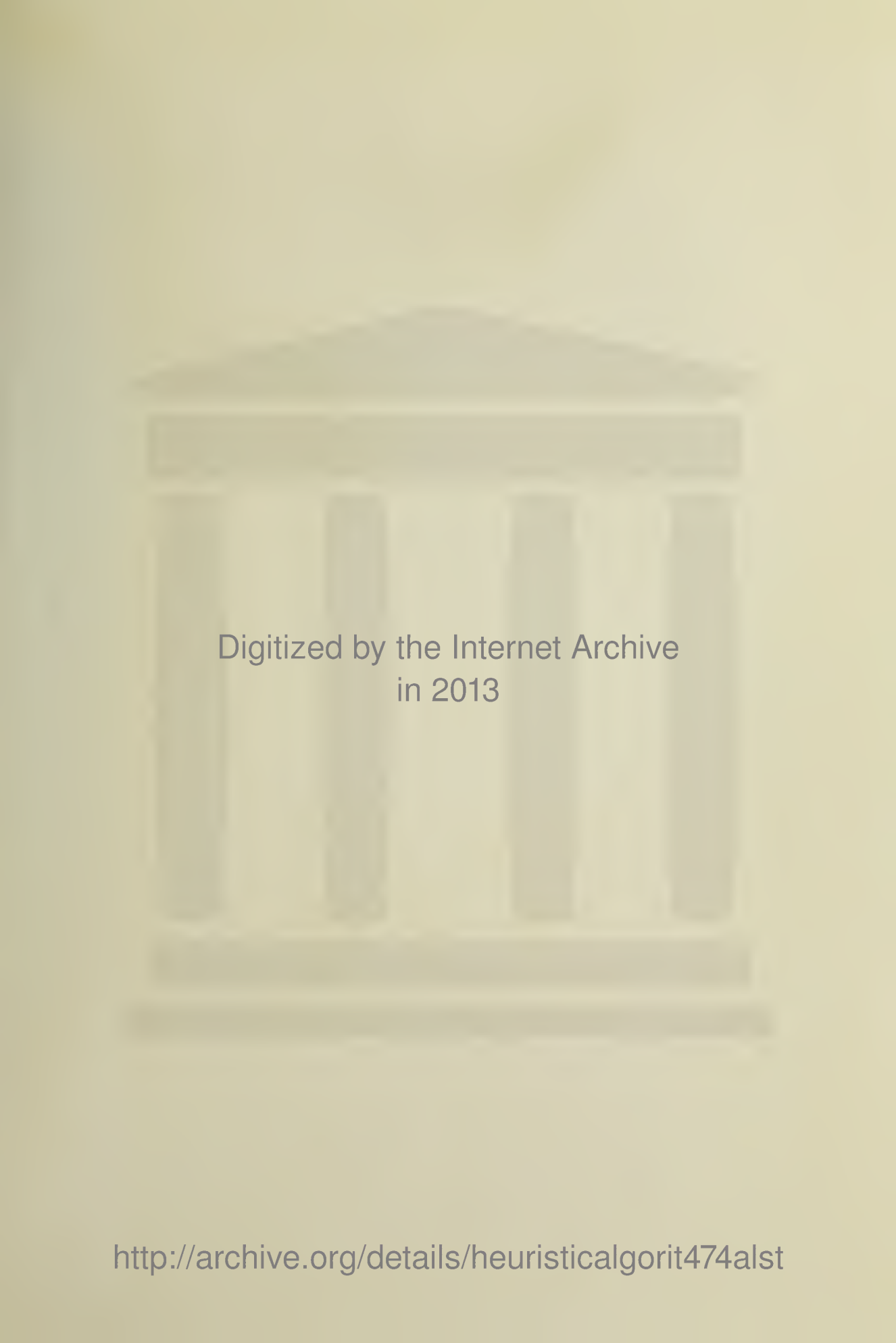


Report No. 474

HEURISTIC ALGORITHMS FOR CONSTRUCTING NEAR-OPTIMAL DECISION TREES

by

Joan Manning Alster

August 17, 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 474

HEURISTIC ALGORITHMS FOR CONSTRUCTING NEAR-OPTIMAL DECISION TREES

by

Joan Manning Alster

August 17, 1971

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, August 1971, and was supported in part by the National Science Foundation and the Department of Computer Science.

ACKNOWLEDGMENT

I would like to express my appreciation to Professor Jurg Nievergelt for the guidance he gave me with this thesis. Also, I would like to thank Professor J. N. Snyder, the University of Illinois Department of Computer Science, and the National Science Foundation for providing the computer time necessary for my research.

TABLE OF CONTENTS

1. INTRODUCTION

The logical structures of certain types of problems may be represented by decision trees. A decision tree is a binary tree whose internal nodes represent points in time at which decisions must be made to take either the left or right branches of these nodes. The root of a decision tree represents the status of a problem before any decisions have been made, and the leaves of the tree represent all possible outcomes of the problem (which the tree represents) which could result from all possible combinations of decisions made at the internal nodes.

There has been a great deal of study done on related tree problems. Two major areas of study have been optimal search trees and Huffman's tree constructions for minimum redundancy codes. Decision trees are, in fact, a generalization of these other types of trees, and there are many optimality problems which arise in the area of decision trees which cannot be solved using the algorithms derived for handling these other, related tree problems.

The problem of decision trees arises in the study of decision tables and in the conversion of limited-entry decision tables to decision trees for the purpose of computer programming. Algorithms have been found for converting a decision table to a computer program which uses a minimum amount of storage. The storage requirement for the program is minimized by using the given decision table to construct a corresponding decision tree which has a minimum number of nodes and

constructing the program from this optimal decision tree. There are also algorithms for converting a decision table to a computer program which executes in a minimum amount of time. The execution time is minimized by assigning a probability, or frequency of occurrence, to each possible outcome (column) in the table and constructing the corresponding decision tree so that the most likely outcomes are resolved at relatively low level nodes of the tree and less likely outcomes are resolved at the higher level nodes of the tree. The decision tree for minimizing execution time will probably contain more than the minimum number of nodes because frequently occurring outcomes will be resolved in as few decision steps (nodes) as possible even if this necessitates additional decision steps for resolving outcomes which seldom occur. [For discussion of the two algorithms mentioned here, see Pollack, "Conversion of Limited Entry Decision Tables to Computer Programs." Communications of the ACM, Vol. 8, No. 11, November 1965, pp. 677-682.]

The decision table application of decision trees is not entirely general. Below is an example of a decision table:

	C_1	C_2		C_n
ρ_1	N	Y	. . .	Y
ρ_2	Y	–	. . .	N
.	.	.	. . .	.
.	.	.	. . .	.
.	.	.	. . .	.
ρ_m	–	N	. . .	Y

Figure 1

A particular case, C_k, is determined by whether each of certain predicates (conditions) $\rho_1, \rho_2, \ldots, \rho_m$ holds (entry in table is Y), does not hold (entry in table is N), or does not apply (entry in table is –). Each –, or "don't-care" entry under a particular case C_k in the table is a substitute for listing two configurations (assignments of Y or N to each ρ_i) of the $\rho_1, \rho_2, \ldots, \rho_m$ in that case (one with a Y where the dash occurs, the other with an N where the dash occurs). Thus, cases containing one or more don't-care entries actually include several configurations of $\rho_1, \ldots, \rho_m$. However, the possible combinations of configurations of $\rho_1, \ldots, \rho_m$ which can be represented by a single column containing don't-care entries is only a subset of all possible combinations of configurations which could be included in a case. There are certain decision tree applications for which it is necessary to be able to assign <u>any</u> combination of configurations of the $\rho_1, \ldots, \rho_m$ to each case (each configurations will belong to only one C_k). Construction of optimal decision trees for each applications cannot be accomplished by using the previously-mentioned algorithms devised for converting decision tables to decision trees.

One such general application of decision trees arises in the problem of trying to optimize the efficiency of branching in a computer program. Consider the following simple example which is illustrated in Figure 2: We are working with the x and y coordinates of points in the Euclidean plane. Assume x and y are non-zero. We wish to branch to different parts of the program depending upon whether we have case 1 (the point is in the first quadrant), case 2 (the point is in the second quadrant), or case 3 (the point is in the third or fourth quadrants).

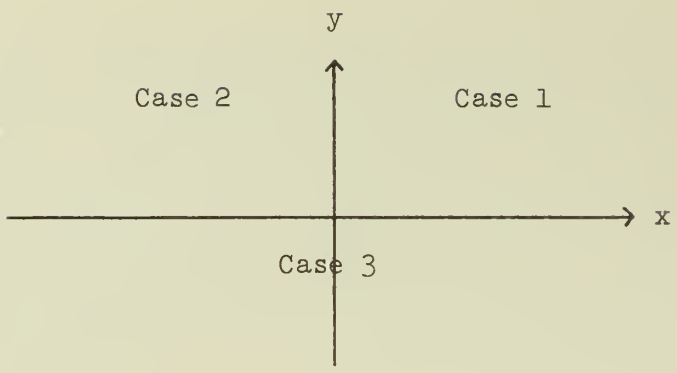


Figure 2

The two possible ways to program this branching process are shown in the two decision trees below.

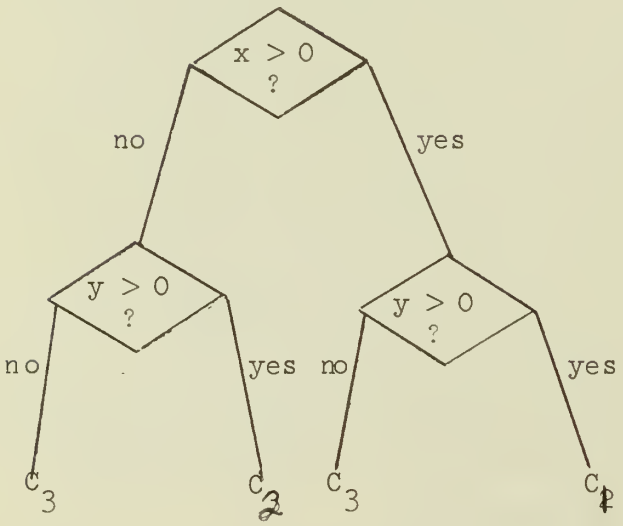


Figure 3a

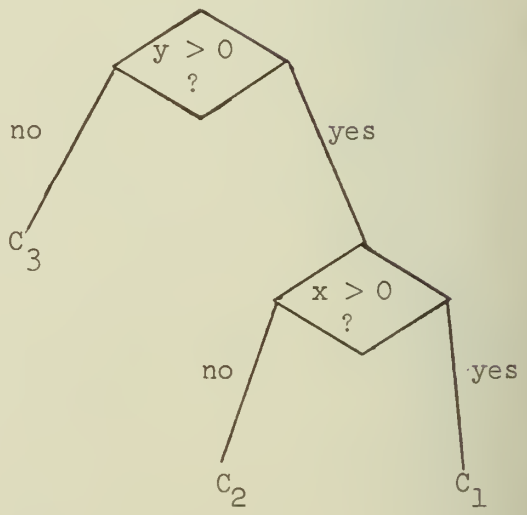


Figure 3b

Since the number of tests required to resolve any case in Figure 3b is always less than or equal to the number of tests required to resolve a case in Figure 3a, Figure 3b represents the preferable programming logic.

For this example, there were only two possible decision trees, so it was convenient to examine both trees and choose the better of the two. However, the relatively complex logic of most problems programmed for the computer makes such a trial-and-error analysis highly impractical. Therefore, we would like to devise a systematic method for analyzing a programming problem and creating a decision tree which will optimize its branching processes. The remainder of this paper will be concerned with this problem.

A programming branching problem can be represented by a table from which a decision tree can be constructed. For example, the problem represented by Figure 2 can be represented by the following table:

	C_1	C_2	C_3	
$\rho_1(x > 0)$	1	0	0	1
$\rho_2(y > 0)$	1	1	0	0

0 = false
1 = true

Figure 4

There are two predicates (conditions), ρ_1 and ρ_2, each of which may be either true or false. Therefore, there are $2^2 = 4$ possible combinations for these two predicates. If ρ_1 and ρ_2 are both true, we have case C_1; if ρ_1 is false and ρ_2 is true, we have case C_2; and if ρ_2 is false (and ρ_1 either true or false), we have case C_3.

If we make use of the "don't-care" symbol in constructing our table, then the table in Figure 4 could be represented as the decision table in Figure 5. (More complicated problems generally will not be representable in decision table form.) However, for the present, we shall choose not to use the "don't-care" representation; therefore, if we have n predicates our table will always contain 2^n columns. This means that if certain combinations of ones and zeroes do not concern us in a particular problem, they still must be entered in the table. However, all these combinations may be grouped into a single C_k, the "else" case.

	C_1	C_2	C_3
ρ_1	1	0	-
ρ_2	1	1	0

Figure 5

Our present study is based on two assumptions: (1) that the cost of making a decision remains constant over all nodes of the decision tree, i.e. the truth value of a particular ρ_i may be determined with equal ease for all ρ_i, and (2) each case occurs with equal probability. When these assumptions hold true, the best branching logic for a computer program will be that for which the corresponding decision tree has a minimal number of nodes. We shall refer to the tree with the minimal number of nodes as the optimal tree. There may be more than one optimal tree for a given problem.

The search for a systematic method for finding the optimal tree has not yet yielded an algorithm which is guaranteed to produce the optimal tree on the first try. However, I have found several heuristic algorithms which, when applied, greatly reduce the effort that is required to produce the optimal tree by trial-and-error methods. These heuristic algorithms will be discussed in Chapters 2 and 3.

Let us first consider the obvious algorithm of finding an optimal decision tree by exhaustive search. Consider the problem illustrated by the following table:

	C_1							C_2			C_3					
ρ_1	0	0	0	0	0	1	1	0	1	1	0	0	1	1	1	1
ρ_2	0	0	1	1	1	0	0	1	0	0	0	0	1	1	1	1
ρ_3	1	1	0	0	1	1	1	1	0	0	0	0	0	0	1	1
ρ_4	0	1	0	1	0	0	1	1	0	1	0	1	0	1	0	1

Figure 6

Here n=4 so we have $2^4=16$ columns in the table. From this table, we shall construct a tree which will have the following structure:

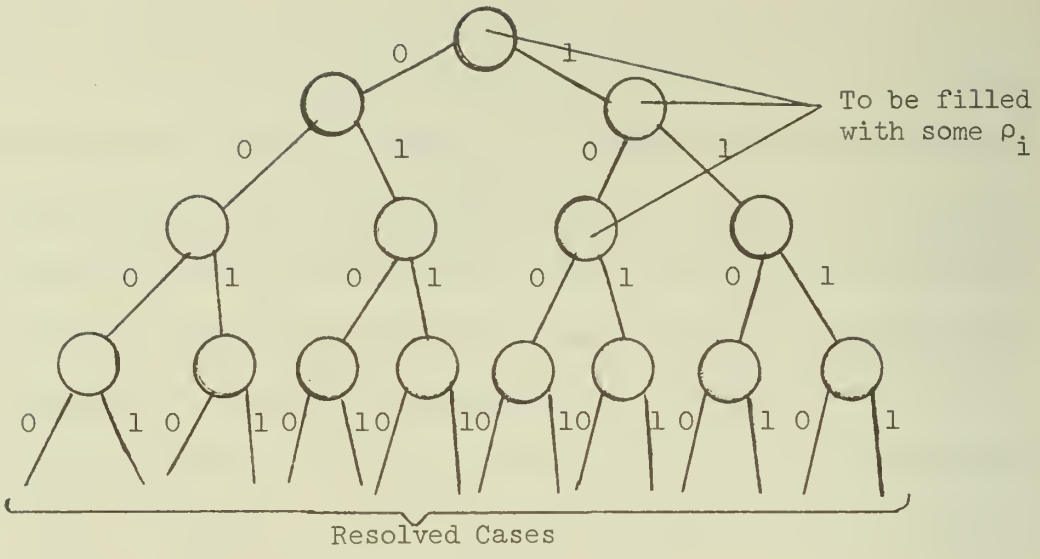


Figure 7

The maximum number of nodes the tree can contain is 15 (= 2^n-1, where n is the number of predicates), but it may contain fewer than 15 nodes if parts of some cases (some columns from the table) can be resolved without testing all of the ρ_i. If we conduct an exhaustive trial-and-error search for the optimal tree, how many trees must we check? Realizing that each root-to-leaf path can contain a particular ρ_i at most once, we see that we have 4 choices for the level zero node, 3^2 choices for the level one nodes, 2^4 choices for the level two nodes, and 1^8 choices for the level three nodes. Thus, the maximum number of trees that would have to be inspected is $4 \cdot 3^2 \cdot 2^4 \cdot 1^8 = 576$ trees. Of course, if some of the trees have fewer than 15 nodes, there will be fewer trees to investigate, but the number will remain quite large--too large, in fact, to make trial-and-error investigation feasible even when n is as small as 4. When the exhaustive trial-and-error search was programmed and run on the computer,

it was found that for this example, 484 trees had to be inspected to discover that there exist two optimal trees with six nodes each. In general, when there are n predicates, the upper bound for the number of trees which must be inspected to ensure obtaining the optimal tree is given by:

Maximum number of trees to inspect =

$$n^{(2^0)} \cdot (n-1)^{(2^1)} \cdot (n-2)^{(2^2)} \cdot \ldots \cdot (n-k)^{(2^k)} \cdot \ldots \cdot (1)^{(2^{(n-1)})}$$

Obviously, an exhaustive trial-and-error search for the optimal decision tree requires too much work to be feasible.

2. HEURISTIC ALGORITHMS

2.1. Algorithm 1 (Constant Cases)

Step 1: To determine which ρ_i to select when constructing the decision tree, look at the table and choose the ρ_i for which the most cases are constant (either all zeroes or all ones), if such a ρ_i exists. For example, in Figure 4, for ρ_1, cases C_1 and C_2 are constant, but for ρ_2, all three cases are constant. Therefore, ρ_2 should be chosen first. Indeed, as shown by Figures 3a and 3b, ρ_2 is the preferable choice.

Step 2: If no such ρ_i exists, proceed as though by the exhaustive search method and apply this "constant case" algorithm to resulting subtables whenever possible. Whenever this algorithm yields a "best" choice for a particular node, no other p_j's need to be tried for that node (unless, of course, in proceeding through the exhaustive search we change a ρ_i which lies on the path between the root and our particular node). Therefore, we eliminate looking at some of the trees we might otherwise have considered when searching exhaustively for the optimal tree. Note that when two or more ρ_i's have the same number of constant cases, each of these ρ_i's should be tried; they might not all prove to be equally good choices. To illustrate Step 2, refer to Figure 6. There are no constant cases for any of the ρ_i's. Therefore, proceed as by the exhaustive search method. Suppose ρ_1 is chosen to be the root of the tree. Construct two subtables:

$\rho_1 = 0$

	C_1					C_2	C_3	
ρ_2	0	0	1	1	1	1	0	0
ρ_3	1	1	0	0	1	1	0	0
ρ_4	0	1	0	1	0	1	0	1

$\rho_1 = 1$

	C_1		C_2		C_3			
ρ_2	0	0	0	0	1	1	1	1
ρ_3	1	1	0	0	0	0	1	1
ρ_4	0	1	0	1	0	1	0	1

Figure 8

To obtain the left descendant of the root ($\rho_1 = 0$), note that both ρ_2 and ρ_3 have two constant cases so both of these predicates must be tried, but ρ_4 need not be tried. To obtain the right descendant of the root ($\rho_1 = 1$), note that ρ_2 has three constant cases, more than either ρ_3 or ρ_4, so choose ρ_2. ρ_3 and ρ_4 never need to be considered. The tree now looks like this:

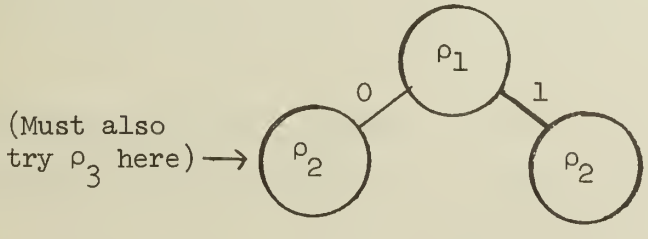


Figure 9

Now construct two more subtables for each bottom node and continue constructing the tree until all the leaves of the tree are resolved cases. After constructing all possible trees with ρ_1 at the root

(eliminating the consideration of some, of course, by using the algorithm), go back and do the same for roots of ρ_2, ρ_3, and ρ_4. (Since the algorithm did not, in this example, yield any information about which ρ_i would be a best choice for the root of the decision tree, all roots must be tried.) The optimal decision tree is the best tree found by the above-described search. When Algorithm 1 was programmed and run on the computer, it was found that for the above example, 36 trees were inspected to find the two optimal trees with six nodes each. This is in contrast to the 484 trees examined in the exhaustive search.

2.2. Algorithm 2 (Weight Function)

Step 1: For each predicate, ρ_i, calculate a weight given by

$$WT_1 = \sum_{\text{all cases}} \sum_{k=1}^{n-1} K \cdot P_K$$

where (a) P_K is the number of groups of 2^K zeroes or ones there are within a case, (b) no zeroes or ones are counted more than once, and (c) longer strings are counted over shorter strings (i.e. four zeroes would be counted as one group of 2^2 zeroes, not two groups of 2^1 zeroes). For example, if one case within a ρ_i contains six zeroes and three ones, the weight for that case would be given by:

$$wt_{case} = \underset{\substack{\text{one group of} \\ 2^2 \text{ zeroes}}}{2 \cdot 1} + \underset{\substack{\text{one group of } 2^1 \text{ zeroes} \\ + \text{ one group of } 2^1 \text{ ones} \\ = \underline{\text{two}} \text{ groups of } 2^1 \\ \text{zeroes or ones}}}{1 \cdot 2} \quad \text{(one 1 left over)}$$

To find WT_1 for ρ_i, sum the weights over all cases.

Step 2: Choose the predicate for which the weight is a maximum. If more than one predicate has the maximum weight, all those with the maximum weight must be tried to ensure that the next node of the decision tree will be filled with the "best" predicate.

We shall evaluate the weights for all the predicates in the example in Figure 6.

$$\rho_1: \quad WT_1 = (2\cdot 1 + 1\cdot 1) + (1\cdot 1) + (2\cdot 1 + 1\cdot 1) = 7$$
$$\rho_2: \quad WT_2 = (2\cdot 1 + 1\cdot 1) + (1\cdot 1) + (2\cdot 1 + 1\cdot 1) = 7$$
$$\rho_3: \quad WT_3 = (2\cdot 1 + 1\cdot 1) + (1\cdot 1) + (2\cdot 1 + 1\cdot 1) = 7$$
$$\rho_4: \quad WT_4 = (2\cdot 1 + 1\cdot 1) + (1\cdot 1) + (1\cdot 2) \qquad = 6$$

The algorithm happens not to be decisive for choosing the root of the decision tree, though it does show that ρ_4 would be the worst choice for the root (so ρ_4 need not be considered as a possible root). When Algorithm 2 was run on the computer, 22 trees were inspected to find the two optimal trees with six nodes each. This is somewhat better than the 36 trees inspected when using Algorithm 1.

2.3. Algorithm 3 (Weight Function)

Step 1: For each predicate, ρ_i, calculate a weight given by

$$WT_2 = \sum_{\text{all cases}} \cdot \sum_{k=2}^{n-1} K^2 \cdot P_K$$

where the notation is the same as that described for Algorithm 2.

Step 2: Same as Step 2 for Algorithm 2.

Again, we shall evaluate the weights for the predicates in Figure 6.

$$\rho_1: \quad (2^2 \cdot 1 + 1^2 \cdot 1) + (1^2 \cdot 1) + (2^2 \cdot 1 + 1^2 \cdot 1) = 11$$
$$\rho_2: \quad (2^2 \cdot 1 + 1^2 \cdot 1) + (1^2 \cdot 1) + (2^2 \cdot 1 + 1^2 \cdot 1) = 11$$
$$\rho_3: \quad (2^2 \cdot 1 + 1^2 \cdot 1) + (1^2 \cdot 1) + (2^2 \cdot 1 + 1^2 \cdot 1) = 11$$
$$\rho_4: \quad (2^2 \cdot 1 + 1^2 \cdot 1) + (1^2 \cdot 1) + (1^2 \cdot 2) = 8$$

Algorithm 3 has the effect of weighing more heavily the larger groups of ones and zeroes than does Algorithm 2. As with the earlier algorithms, Algorithm 3 is indecisive for selecting a root in the example shown here. However, when this algorithm was run on a computer, it was found that only eight trees needed to be investigated to find the two optimal trees with six nodes each. This is a considerable improvement over the 22 trees which had to be investigated when Algorithm 2 was used.

The three algorithms plus the exhaustive search were programmed for the computer and run for eight different problem situations, each with $n=4$ (hand testing is fairly easy for $n \leq 3$). The eight trials were not random problem situations, but rather, were carefully selected to represent as wide a range of different types of situations as possible. The number of cases varied from two to five. The results are summarized below:

		EXHAUSTIVE SEARCH			ALGORITHM 1 (CONSTANT CASES)			ALGORITHM 2 (WEIGHT FUNCTION)			ALGORITHM 3 (WEIGHT FUNCTION)		
Trial	Number of Cases	Number of Optimal Trees Found	Number of Nodes in Optimal Tree	Number of Trees Inspected	Number of Optimal Trees Found	Number of Nodes in Optimal Tree	Number of Trees Inspected	Number of Optimal Trees Found	Number of Nodes in Optimal Tree	Number of Trees Inspected	Number of Optimal Trees Found	Number of Nodes in Optimal Tree	Number of Trees Inspected
1	3	2	6	484	2	6	36	2	6	22	2	6	8
2	4	3	8	480	4	9*	30	3	8	18	4	9*	8
3	5	576	15	576	576	15	576	576	15	576	576	15	576
4	4	2	6	312	2	6	8	2	6	6	2	6	6
5	2	2	3	450	2	3	74	2	3	90	2	3	74
6	5	2	4	468	2	4	46	2	4	16	2	4	2
7	5	31	11	576	12	11	72	1	11	25	16	13*	16
8	4	20	11	576	7	11	128	18	11	82	12	11	36

Figure 10

*Instances in which an algorithm failed to find an optimal tree.

As the table shows, the algorithms did not always yield all the optimal trees. However, that is not a problem because any optimal tree is considered to be as good as any other. Algorithm 2 nearly always required examination of fewer trees than Algorithm 1, and Algorithm 3 was always at least as good, and often better than Algorithm 2 in this respect. The starred (*) entries in the table indicate instances in which an algorithm failed to find an optimal tree. If one is looking for accuracy in finding the optimal tree, Algorithm 2 appears best. However, if one is more interested in speed and cares only that he find a "good" (and not necessarily a "best") tree, Algorithm 3 is preferable. In any case, all the algorithms reduced substantially the exhaustive search effort, and all found good, though not always optimal, trees. The algorithms remain untested for $n > 4$.

3. AN IMPROVED HEURISTIC ALGORITHM

After extensive experimentation with Algorithms 1, 2, and 3, I became convinced that the single most important consideration when selecting a "best" predicate is the number of occurrences of zeroes or ones in groups of powers of two within cases. This suggests considering decision tables which contain "don't-care" (-) entries since the dashes are helpful in indicating where powers of two occur. For example, the case C_k in Figure 11a could be represented as in Figure 11b with the dash (-) signifying that ρ_1 is an undesirable choice as far as case C_k is concerned.

	C_k	
ρ_1	0	1
ρ_2	1	1
ρ_3	1	1
ρ_4	0	0

Figure 11a

	C_k
ρ_1	-
ρ_2	1
ρ_3	1
ρ_4	0

Figure 11b

The Quine-McCluskey minimization procedure is used to construct the table with "don't-care" entries. For $n \leq 4$, a Karnaugh map is also useful. Let us use the Quine-McCluskey procedure to convert the table in Figure 6 to a table containing "don't-care" entries. Each case must be handled separately, so we begin with C_1. First, we find the prime implicants as shown below:

Decimal Representation of Binary Sequence	Complete Sequences of ρ_i's in Case 1	Derived "Don't-Care" Sequences	
(2)	0 0 1 0 √	(2,3): 0 0 1 - √	(2,3,10,11): - 0 1 -
(4)	0 1 0 0 √	(2,6): 0 - 1 0	
(3)	0 0 1 1 √	(2,10): - 0 1 0 √	
(5)	0 1 0 1 √	(4,5): 0 1 0 -	
(6)	0 1 1 0 √	(4,6): 0 1 - 0	
(10)	1 0 1 0 √	(3,11): - 0 1 1 √	
(11)	1 0 1 1	(10,11): 1 0 1 - √	

Prime Implicants (non-checked sequences)

- 0 1 -
0 - 1 0
0 1 0 -
0 1 - 0

Figure 12

The following comments are made in regard to Figure 12:

(1) The columns in Case C_1 in Figure 6 are represented horizontally as complete sequences of ρ_i's.

(2) To facilitate finding derived sequences, the complete sequences (and thus, the derived sequences also) are listed in order of the number of 1's they contain.

(3) In order for two sequences to be combined, they must be identical in all but the one position in which one sequence will contain a 1, the other a 0.

(4) When two sequences are combined, they are checked ($\surd$) in the table. The prime implicants are all those sequences which are unchecked after no more sequences can be combined to form more derived sequences.

(5) Every sequence must be compared with all sequences below it which contain the same number of 1's or one more 1 than the given sequence (even if some of these sequences are already checked), and all possible derived sequences must be written.

After the prime implicants have been obtained, a McCluskey Chart is constructed. For our example, the following chart is obtained.

Derived Don't-Care Sequences	Derived From Which Complete Sequences	Sequences Belonging to Case C_1						
		2	3	4	5	6	10	11
- 0 1 -	(2,3,10,11)	X	X				X	X
0 - 1 0	(2,6)	X				X		
0 1 0 -	(4,5)			X	X			
0 1 - 0	(4,6)			X		X		

Essential Prime Implicants: - 0 1 -

0 1 0 -

Non-Essential Prime Implicants: 0 - 1 0 } Must have one of these to cover "6" column

0 1 - 0 }

Figure 13

In the new table containing "don't-care" entries, C_1 is represented as follows:

	C_1				
ρ_1	-	0	0	or	0
ρ_2	0	1	-		1
ρ_3	1	0	1		-
ρ_4	-	-	0		0

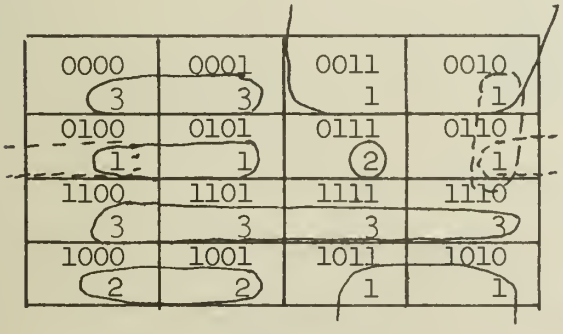
Figure 14

The table entries for C_2 and C_3 can also be derived using the Quine-McCluskey procedure to yield the following table:

	C_1				C_2	C_3
ρ_1	- 0	0		0	1 0	1 0
ρ_2	0 1	-	or	1	0 1	1 0
ρ_3	1 0	1		-	0 1	- 0
ρ_4	- -	0		0	- 1	- -

Figure 15

Figure 15 could also have been all or partly derived from the following Karnaugh map which can be obtained directly from Figure 6. Of course, use of a Karnaugh map is only practical for $n \leq 4$.



Entries in map indicate to which case the corresponding sequence of 1's and 0's belongs.

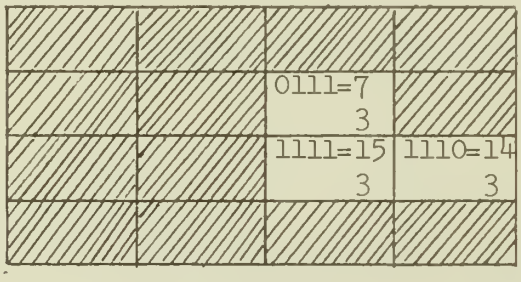
⬭ = Essential Prime Implicant

⬭ = Non-Essential Prime Implicant

Figure 16

The following comments are in regard to Figure 15:

(1) Essential prime implicants are represented without brackets ([]) with one important exception: when two essential prime implicants overlap in all but one column of the McCluskey Chart or in all but one box of a Karnaugh map (that one column or box being the one which makes each of the two essential prime implicants essential), then when entering the two essential prime implicants in the table, put brackets around each and insert the word "or" between the entries. For example, suppose the following map and chart illustrate a case $C_3(n=4)$:



Karnaugh Map

	7	14	15
(7,15): -111	x		x
(14,15): 111-		x	x

McCluskey Chart

Figure 17

Both -111 and 111- are essential prime implicants and they overlap in all but one box in the Karnaugh map and all but one column of the McCluskey Chart (-111 is essential because it contains "7", 111- because it contains "14"). Therefore, the case C_3 would be represented:

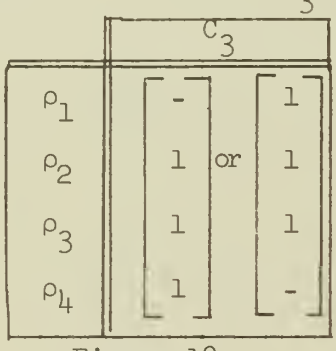


Figure 18

(2) When there is a choice of which non-essential prime implicant (or combination of non-essential prime implicants) to select, list all choices in the table, each in brackets, and join the brackets by the word "or". Call such a group of bracketed entries joined by the word "or" an or-group (each bracketed entry will be called a component of the or-group), and define the or-number of an or-group to be the number of components contained in the or-group. There may be more than one or-group in a case. For example, a case could have the following structure: []or[] []or[]or[]. The first or-group has an or-number of 2, the second an or-number of 3.

(3) Define the case-count for a case to be 2^r where

$$r = \text{(number of dashes occurring in non-bracketed entries in the case)} + \sum_{\substack{\text{all or-groups} \\ \text{in the case}}} \text{(maximum number of dashes in any component of the or-group)}$$

(4) Define the dash-count for a particular ρ_i to be the sum of the case-counts corresponding to each non-bracketed dash in the row plus the sum of $\left(\frac{1}{\text{or-number}}\right) \cdot$ (corresponding case-count) for each bracketed dash in the row.

The preceding comments and definitions lead to the statement of Algorithm 4 for constructing an optimal decision tree. The notions of case-count and dash-count resemble those described by Pollack in his article, "Conversion of Limited-Entry Decision Tables to Computer Programs" (cited earlier in this paper).

3.1. Algorithm 4 (Dash-Count)

Step 1: Using a Karnaugh map or the Quine-McCluskey procedure plus the bracketing rules described above, construct a table containing "don't-care" entries.

Step 2: Compute the case-count for each case.

Step 3: Compute the dash-count for each ρ_i.

Step 4: Select the ρ_i for which the dash-count is a minimum. If more than one ρ_i has minimum dash-count, select any of the ρ_i's with a minimum dash-count.

The case-counts and dash-counts for Figure 15 are shown below:

Case Count	$2^{3+1} = 2^4 = 16$				$2^1 = 2$	$2^3 = 8$	
	C_1				C_2	C_3	Dash Counts
ρ_1	-	0	0	0	1 0	1 0	16
ρ_2	0	1	-	or 1	0 1	1 0	(1/2)(16) = 8
ρ_3	1	0	1	-	0 1	- 0	(1/2)(16)+8 = 16
ρ_4	-	-	0	0	- 1	- -	2(16)+2+2(8) = 50

Figure 19

Since ρ_2 has the minimum dash count, choose ρ_2 to be the root (ρ_2 is indeed the root of both optimal trees). Use the original table (not the one with the "don't-care" entries) to create two subtables and apply Algorithm 4 to each of the subtables.

One very significant advantage of Algorithm 4 over the other algorithms is that cases which are of no concern in a particular problem can be designated as such (as a "d" in a Karnaugh map or Quine-McCluskey chart) rather than having to be combined into a single "else" case. For example, suppose we have a problem in which n = 4 and in which we assign only 13 predicate sequences to cases C_1 through C_4. We have no concern for what happens to the three sequences 0010, 0011, and 0111. In order to apply our earlier algorithms, we would have to first combine these three "else" sequences into a single case--C_5. However, for Algorithm 4, this "else" case is not necessary. Suppose the original table is:

	C_1				C_2			C_3		C_4			
ρ_1	1	0	0	0	0	1	1	1	1	1	1	0	1
ρ_2	1	1	0	1	0	0	0	0	1	0	1	1	1
ρ_3	0	0	0	0	0	0	1	1	1	0	0	1	1
ρ_4	1	1	0	0	1	1	1	0	0	0	0	1	1

Figure 20

The following Karnaugh map can be constructed from this table:

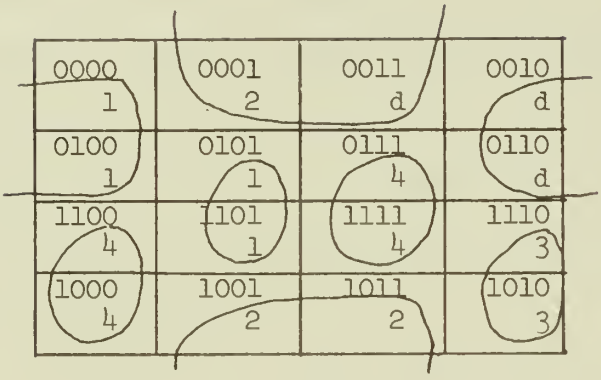
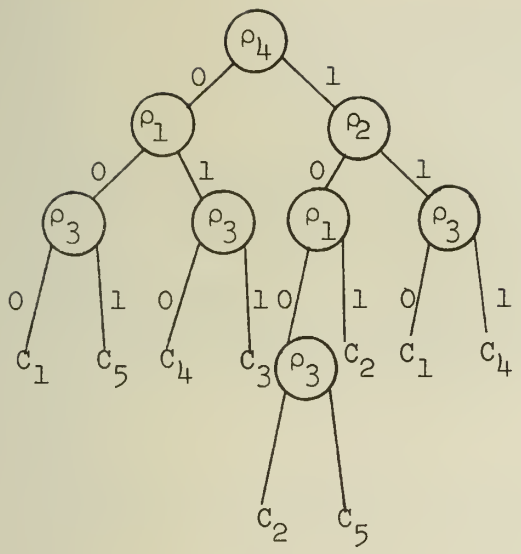
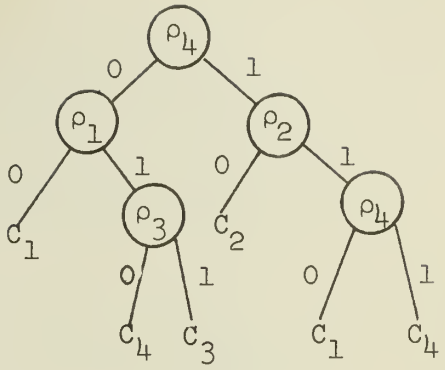


Figure 21

The "d" entries provide a far more accurate representation of the problem than would a fifth, "else" case; and as the Karnaugh map indicates, a far better optimal tree will result when the "d" entries are used. Figure 22 shows an optimal decision tree obtained using an "else" case and one obtained using "d" entries.



Optimal Decision Tree Using C_5 as "Else" Case (8 nodes)



Optimal Decision Tree Using "d" Entries (5 nodes)

Figure 22

Algorithm 4 has yielded an optimal decision tree in many examples on which it was tested. However, Toshio Yasui (Ph.D. student, University of Illinois) has found the following counterexample to Algorithm 4. Given the decision table in Figure 23,

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
ρ_1	–	0	1	–	0	1	1	0	0
ρ_2	0	–	1	0	–	1	0	1	1
ρ_3	0	1	–	0	1	–	1	0	0
ρ_4	0	0	0	1	1	1	–	0	1

Figure 23

Algorithm 4 indicates that the optimal tree would have ρ_4 for its root. In fact, however, all the optimal trees have either ρ_1, ρ_2, or ρ_3 for their roots with a root of ρ_4 yielding no optimal trees. Therefore, although Algorithm 4 provides an efficient, systematic method for finding a very "good" decision tree, it will, in some situations, fail to yield the optimal decision tree. Algorithm 4 is judged, however, to be generally more reliable than the algorithms presented earlier in this paper.

LIST OF REFERENCES

Pollack, Solomon, L., "Conversion of Limited Entry Decision Tables to Computer Programs," Communications of the ACM, Vol. 8, No. 11, November 1965, pp. 677-682.

Yasui, Toshio, "Some Combinatorial Aspects of Decision Table Optimization Problems," Ph.D. Dissertation (in progress), University of Illinois at Urbana-Champaign, Urbana, Illinois.